Introduction to subversion
version control
—track files over a period of time or by what developers
A repository(central location) with revision numbers.

```
$ Command [arguments] [options]
java        %java compiler
java -version
```

Java Fundamentals @codecademy.com
※ data type:
①int ∈[－2147483648, 231-1=2147483648]
②boolean ∈{true, false} ={0,1}
③char  represent single characters  ‘c’

※ Variable:   int var=1;

※ Whitespace: helps make code easy to read for you and others.

※ comment:
single line comment //
multi-line comment /* */

arithmetic operator:  +(add) -(subtract) *(multiply) /(divide) %(reminder)
relational operator: <  <= > >=
equality operator: == !=
1.conditionals and control flow:
Control flow allows Java programs to execute code blocks depending on Boolean expressions.
※ Boolean operator: and(&&)  or (II)  not(!)
The precedence of each Boolean operator is as follows:
! is evaluated first  && is evaluated second  II is evaluated third
※ if statement
※ ternary conditional statement: write if/else statements in a single line of code
char result= (score > 60) ? 'P' : ‘F’;
※ switch statement

2.object-oriented programming:
Java is an object-oriented programming (OOP) language, which means that we can design classes, objects, and methods that can perform certain actions.
Class: a blueprint for how a data structure should function
Constructor: instructs the class to set up the initial state of an object
Object: instance of a class that stores the state of a class
Method: set of instructions that can be called on an object
Parameter: values that can be specified when creating an object or calling a method
Return value: specifies the data type that a method will return after it runs
Inheritance: allows one class to use functionality defined in another class

3.data structure
for loop:     for(initialization; test condition; increment) { statement; }
ArrayList: stores a list of data of a specified type (a pre-defined Java class)
for each loop:
HashMap: contains a set of keys and a value for each key

# Software Design

An introduction to software design and development concepts, methods, and tools using a statically-typed object-oriented programming language such as Java.
Topics from: version control, unit testing, refactoring, object-oriented design and development, design patterns, advanced IDE usage, regular expressions, and reflection. Representation of floating-point numbers and introduction to numerical computation.

—Intro to Java (types, inheritance, generics, etc.)
—Topics related to OOD (design patterns, CRC, UML, etc.)
—Tips for working on complicated programs (version control (git), floating-point issues, regular expressions, etc.)

What we want you to Learn:
—Fundamental code development techniques used professionally:
version control (using git)
aspects of team dynamics
design patterns
tools for designing and analyzing programs
a large Integrated Development Environment (Eclipse)
the command line
—Object-oriented programming:
strong typing
lots of inheritance
more unit testing
a memory model for Java
exception handling
floating-point issues

## Good Design

(features of program that you decide before any code is written)
—how many classes comprise your program
—whether or not a class should have subclasses
—use interfaces or abstract classes
—should one class contain certain methods or should they be broken up into multiple different classes?

## SOLID Principle

A SOLID understanding of Objected-Oriented Design
1.S = single responsibility principle (do one thing and do it well)
Every class should have a single responsibility/service; responsibility should be entirely encapsulated by the class; all class services should be aligned with that responsibility; makes the class more robust and more reusable.
2.O = software entities(class/module/function etc.) should be Open for extension (how code changes program), but Closed for modification (how program changes programs)
good design makes everything as private as possible.
Add new features not by modifying the original class, but rather by extending it and adding new behaviours; the derived class may or may not have the same interface as the original class.
3.L = Liskov Substitution Principle (objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program)
If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
S is a subtype of T: s is a child class of T or s implements interface T
If C is a child class of P, then we should be able to substitute C for P in our code without breaking it.
The sub classes should only extend (add behaviours), not modify or remove them.
4.I = Interface Segregation Principle (many client-specific interfaces are better than one general-purpose interface)
No client should be forced to depend on methods it does not use; better to have lots of small, specific interfaces than fewer larger ones; easier to extend and modify the design.
5.D = Dependency Inversion Principle (depend upon abstractions, [not] concretions)
abstraction layer between low-level classes and high-level classes;
High-level modules should not depend on low-level modules; both should depend on abstraction.

Abstractions should not depend on details; details should depend on abstractions.

## Good Style

(features that you create while coding)
enough comments so that running Javadoc will create an easily understandable summary of your program
not too many comments that the code is difficult to read
using the Java naming conventions for classes, variables, and methods

## Java Naming Conventions

Java Language Specification recommends: Use camelCase not pothole_case.
**Class**: a noun phrase (describes the class accurately), starting with a Capital
(UserAccount, RoomBookManager)
**Method**: a verb phrase (describe what the method does), starting with lower case
(getName, setFrequency)
Instance **Variable**: a noun phrase (describe the role of the value) starting with a lower case letter
(x describes an x-coordinate, numChairs describes the number of chairs)
**Constant**: all UPPER_CASE, pothole
(MAX_ENROLMENT)

Everything start with lower case except objects.

## Week 1

## Define Class in Java

```
public class Circle {
        private String radius;
}
```

Java is a strongly-typed language.
radius is an **instance variable**. Each object/instance of the Circle class has its own radius variable.

Main Component of class: field/variable, method, constructor

**Constructors**:
constructor和class同名，没有return type
constructor call会initiate所有instance variable
class中如果没有constructor, 会自动生成一个无参的constructor
constructor中第一句默认call super的无参constructor
如果super中没有无参的constructor，会CompileError; 需要自己declare call to super “super();”
A constructor has the same name as the class, no return type (not even void)

A class can have multiple constructors, as long as their signatures are different.
If you define no constructors, the compiler supplies one with no parameters and no body.
If you define any constructor for a class, the compiler will no longer supply the default constructor.

**Variable Declaration**

```
Circle C1           =            new Circle();
```
type  variable name                    new: i am calling the constructor next

```
Object a            =            new String(“Hello world”);
```
reference type: 当提到变量 a时, assume他是一个Object
actual type: 实际上a是一个String

```
private String radius = new String();
String radius “three”;                       // this only works for String

public class Circle {
        private int radius;
        private String name = “Happy Fun Circle”;
        private char a;

        public Circle() {
        }

        public Circle(int radius, char alpha) {        // int is primitive variable (start with lower case)
                this.radius = radius;                  // this is similar to self in Python
                a = alpha;                 // there is only one a, so no ambiguity  (can use this.a as well)
        }

        public Circle(int radius) {                    // int is primitive variable (start with lower case)
                this.radius = radius;                  // this is similar to self in Python
        }
}
```

**this**: an instance variable that you get without declaring it (like self in Python), its value is the address of object whose method has been called.

Define Methods:
A method must have a return type declared. Use void if nothing is returned. (return expression;)
If the expression is omitted or if the end of the method is reached without executing a return statement, nothing is returned.

## Parameters

When passing an argument to a method, you pass what is in the variable’s box.
—For class types (object), you are passing a reference. (like in Python)
—For primitive types, you are passing a value. (Python cannot do anything like this)

8 Primitive Types

| Primitive Type | Wrapper Class | Initial Value |
|---|---|---|
| byte | Byte | 0 |
| short | Short | 0 |
| long | Long | 0L |
| int | Integer | 0 |
| float | Float | 0.0f |
| double | Double | 0.0d |
| char | Character | u0000’ |
| boolean | Boolean | FALSE |

String is not primitive type.
using == to compare primitive type, using equals() to compare Objects; using == compares Object memory address.

**Autoboxing & Unboxing**
Integer is the class associated with the primitive type int.
When use == operator on primitive types, Java compares the values of the primitives.
For subclasses of Object, == compares the memory address that each instance points to.

(有些情况下, compiler会自动在primitive和wrapper class之间切换)
a.用 == 比较, 会自动把wrapper的Object auto unboxing.

```
int x = 5;
Integer y = new Integer(5);
System.out.println(x == y);             // true (unboxing)
```

b.用equals比较, 会自动把primitive的autoboxing.
根据变量需求自动autoboxing / unboxing.

```
public class Trivial {
        public void static log(Integer a);
}
Trivial.log(123);
```

## Encapsulation

在class定义中，一般内容都是不面向public的
提供public interface/methods给外界使用
外界通过getter/setter来访问内部变量(Reflection)

class —provide an abstraction or a service
provide access to information through a well-defined interface (public methods of the class) and hide the implementation details
We can change the implementation (to improve speed, reliability or readability) and no other code has to change.

Make all non-final instance variable either private(accessible only within the class) or protected (accessible only within the package). When desired, give outside access using getter and setter methods.

A final variable cannot change value; it is a constant.

## Accessibility (MUST SPECIFY)

Access Modifier: public -> protected -> default (package private) -> private
class -> package -> subclass -> world
public are callable from anywhere, private are callable only from this class.
Variables declared in a method are local to that method.

Note: can create private constructor if only want to use the constructor inside this class

| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| Public | Yes | Yes | Yes | Yes |
| Protected | Yes | Yes | Yes | No |
| Default (package private) | Yes | Yes | No | No |
| Private | Yes | No | No | No |

```
        private Circle() {
        }

        public static Circle getCircle() {                          // static
                return new Circle();
        }

        public String getName() {
                return name;
        }

        public void setRadius(int radius) {                         // void, since no return type
                this.radius = radius;
        }


public class Demo {
        public static void main(Syring[] args) {
                Circle c1 = Circle.getCircle();
                String s = c1.getName();                    // ok! return type match

                String s2 = 5;                              // Error

                System.out println

        }
}
```

```
public class ClassName {
        class variables
        constructors
        methods
}
```

| Primitive: | Object Type: | (primitive are only things that are not objects) |
|---|---|---|
| int | Integer | |
| float | Float | |
| char | Char | |
| double | | |

Wrapper Classes ????????

—Compile (are instructions ok? if yes, —> bit (bike?) code)
compile error: types are wrong;
—Run

Lab 1: IDE IntelliJ

—folder to store projects
—project File ~ New ~ Project
Project SDK Java 1.8 /local/packages/jdk1.8.0
—packages New ~ Package ~ src
—Java Class New ~ Java Class

Review: primitive types & objects

## Week 2

### Javadoc ( )

Like a Python docstring, but more structured and placed above the method.
Javadoc is written for classes, member variables and member methods.
This is where the Java API documentation comes from.

folders (packages) for user interface, database …
organize information in different folders

**Static**: anything that can be used without creating an instance
static的method可以直接使用class名字进行调用，不需要instance
static的variable是class level的，这个variable被所有instances share

Object: at the top of the entire hierarchy

http://docs.oracle.com/javase/tutorial/java/TOC.html
https://www.sololearn.com/Course/Java/

### Running Programs

To run a program, it must be translated from the high-level programming language it is written in to a low-level machine language whose instructions can be executed.
2 flavours of translation:
①Interpretation: translate and execute one statement at a time. ~Python
②Compilation: translate the entire program (once), then execute (any number of times) ~C
③Hybrid:

translate to something intermediate (in Java, bytecode), Java Virtual Machine runs this intermediate code ~Java

Compiling Java (Need to compile, the run)

| | |
|---|---|
| $ javac project/Demo.java | compile using javac |
| $ java project.Demo | run the program using java |

Classes are organized into packages.

[Example]:
—If your program has only one package (called "*project*" for example), then you include the line "**package project;**" at the top of each file to import the package.
Then put all of the files in a folder called "project". In other words, the folder should have the same name as the package.
To run your program, navigate to the folder that contains project. Then type:

```
$ javac project/Demo.java          $ java project.Demo
```

This example assumes that the package is called "project" and that the main method is in a file called Demo.java (i.e. in the Demo class)

—If your program has more than one package, make sure that the class containing the main method includes "**import packageName;**" for each package in the program.
All packages should be located in the same folder. For example, you might have a folder called "csc207" that contains package "project" and also a few other packages that are imported into files inside "project".

## Inheritance

关键词为extends, 只能extends一个; 自己有,
用自己的; 自己没有, 向上找。 (method is inherited from parent class, and called in parent class block)
注意@Override等annotation
Call super的方式为super.xxx

All classes form a tree called inheritance hierarchy with Object at the root.
Object is the root of the entire hierarchy tree in Java.
Class Object does not have a parent. All other Java classes have one parent.
If a class has no parent declared, it is a child of class Object.
A parent class can have multiple child classes.
Class Object guarantees that every class inherits methods toString, equals and others…

Inheritance allows one class to inherit the data and methods of another class.
In a subclass, *super* refers to the part of the object defined by the parent class.

| | |
|---|---|
| super.attribute | refer to an attribute (data member or method) in the parent class |
| super(arguments) | call a constructor defined in the parent class |

If the first step of a constructor is super(arguments), the appropriate constructor in the parent class is called.
Otherwise, the no-argument constructor in the parent is called.
Net effect on order, if, say, A is parent of B is parent of C?

[Example]:
Suppose class Child extends class Parent. An instance of Child has
• a Child part, with all the data members and methods of Child
• a Parent part, with all the data members and methods of Parent
• a Grandparent part, ... etc., all the way up to Object.
An instance of Child can be used anywhere that a Parent is legal.
• But not the other way around.

## Name Look-Up Rules: Override & Shadow/Hide

For a method call expression.method(arguments)
—Java looks for method in the most specific, bottom-most part of the object referred to by expression.
—If it is not defined there, Java looks upward until it is found; else it is an error.
For a reference to an instance variable expression.variable
—Java determines the type of expression and looks in that box
—If it is not defined there, Java looks upward until it is found; else it is an error.

| | **Static** | **Instance** |
|---|---|---|
| **Variable** | shadow | shadow |
| **Method** | shadow | override |

static method (with static)
instance method (w/o static)

instance primitive (int i = 3;)

[Example]:
Suppose class A and its subclass AChild each have an instance variable x and an instance method m.
A's m is overridden by AChild's m (this is often a good idea, we often want to specialize behaviour in a subclass)
A's x is shadowed by A_Child's x (this is confusing and rarely a good idea)

If a method must not be overridden in a descendant, declare it final.

Sub sub = new Sub();
type                constructor

static method shadow (use .. one)
but instance method override

[Example]:
①Static Methods and Static Variable:

```
Sup.staticMethod();              // Super's
Sub.staticMethod();              // Sub's
```

②Instance Method:

```
Sup sup = new Sup();
Sub sub = new Sub();
Sup supsub = new Sub();

sup.instanceMethod();                    // Super's
sub.instanceMethod();                    // Sub's
((Sup)sub).instanceMethod();             // Sub's
supsub.instanceMethod();                 // Sub's

Note: if there is a Sup & a Sub, instance method always use the bottom one. (Override)
For static method/variable and instance variable, can cast down. NEVER can cast up!
```

We can cast to access a shadowed instance variable, static variable or static method, but we cannot access a shadowed instance method.

```
((Sup)sub).instanceMethod();             // Sub's instance method
```

③Static Methods through the instances:

```
sup.staticMethod();                              // Super's
sub.staticMethod();                              // Sub's
((Sup) sub).staticMethod();                      // Sup's static method
supsub.staticMethod();                           // Super's
```

④static and instance variables:

```
sup.i                      sup.si                      // Super's
```

```
sub.i                      sub.si                     // Sub's
supsub.i                   supsub.si                  // Super's
(Sub)supsub).i             (Sub)supsub).si            // Sub's
```

Instances Method:
class A has method aMethod();
class B has method bMethod();

B extends A

A x = new B(); // it will call super() automatically
A x2 = new A(); // no bMethod for x2 at all

x.aMethod(); // allowed
x.bMethod(); // not allowed !!!
((B)x).bMethod(); // cast
x2.bMethod(); // will not run!

Object s = new String("Hello");
s.charAt(0); // will not run

Static Method: call the static method statically. (call them by the class name, not through instances)
A.staticAMethod();
B.staticBMethod;

x.staticMethod(); // will always use the parent class version
Shadow—use the parent version
Override—use the child version

## Casting (the type of instance)

((Student) q).toString() to get the toString() method of class Student

```
Object o = new String("hello");          // allowed, because String is a subclass of Object
char c = o.charAt(1);
// if we could run this code, Java would find the charAt method in o, since it refers to a String object;
// But the code won't compile, because the compiler cannot be sure it will find the charAt method in o,
// Compiler does not run the code, it can only look at the type of o
char c = ((String) o).charAt(1);         // cast o as String
```
Person q = new Student(); // q is a person

Casting does not change the reference type of instance permanently.

```
public class Test {
    public static void method(Object t) {  System.out.println("class Object");}
    public static void method(Test t) {  System.out.println("class Test");}
    public static void main(String[] args) {
        Test t = new Test();
        method(t);               // class Test
        method((Object) t);      // class Object
        method(t);               // class Test
    }
}
```

Java Platform SE 8

Note: static method is really hard to test. there is no way to encapsulated it. so try not make too many static method in code!!!

**Direct Initialization of Instance Variables**
You can initialize instance variables inside constructor(s).

Alternative: initialize in the same statement where they are declared.
Limitations:

can only refer to variables that have been initialized in previous lines.
can only use a single expression to compute the initial value.

What happens when Call the Object??

①Allocate memory for the new object

②Initialize the instance variables to their default values

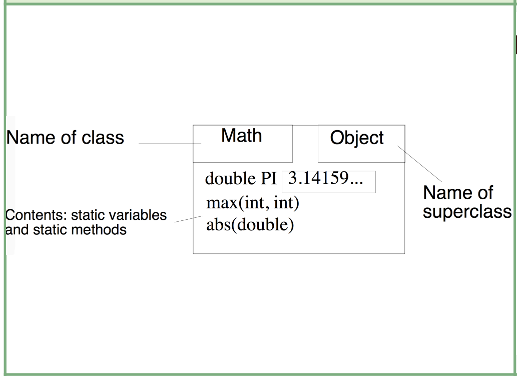
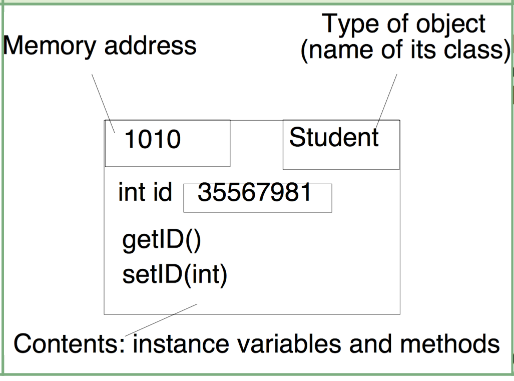
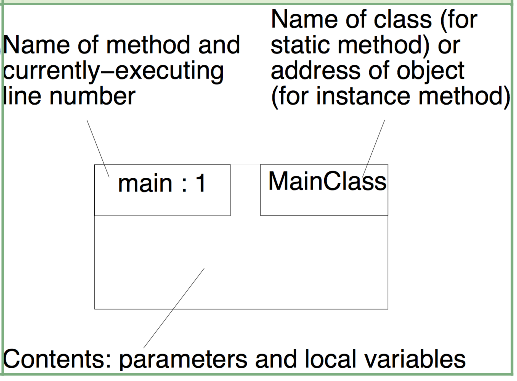
0 for ints, false for booleans.. null for class types

③Call the appropriate constructor in the parent class

The one called on the first line, if the first line is super(arguments), else the non-argument constructor

④Execute any direct initializations in the order in which they occur.

⑤Execute the rest of the constructor

| Static Box | Instance | Method Frame |
|---|---|---|
|  Name of class<br>Math<br>Object<br>double PI 3.14159...<br>max(int, int)<br>abs(double)<br>Contents: static variables and static methods<br>Name of superclass |  Memory address<br>Type of object (name of its class)<br>1010<br>Student<br>int id 35567981<br>getID()<br>setID(int)<br>Contents: instance variables and methods |  Name of method and currently-executing line number<br>Name of class (for static method) or address of object (for instance method)<br>main : 1<br>MainClass<br>Contents: parameters and local variables |

## Week 3

hashcode method (ignore)
equal (same name? same id? or same id and same name)
o1 is an instance person, it cannot be null, o2 can be null.

```
$ javadoc **/java
```
(check html files, to know what is in package)

if (this == obj)
neither is primitive, compare address; if primitive, compare value.
if not, then maybe two same instances but store in different address, so check more.

```
$ javac university/UofT.java
$ java university.UofT
```
(compile to get bitcode)
(run bitcode)

public abstract class .. {}  // just signature

abstract: has at least one method that we don't want to implement.
abstract — no body,
abstract ~ decide the method should be there. when write subclasses to implement.

@override (tell the compiler to check if there is an existing method with the same name, "Am I overriding?")

Once we return a value, the method is done.

Every class can have at most one parent class. (otherwise, conflict!)

## Abstraction

class上要加keyword abstract

只要有abstract method存在, class就是abstract的

abstract class同样可以继承，abstract class可以没有abstract method, abstract method没有body,

abstract class 不能被instantiate

A class may define methods without giving a body.
Each of those methods must be declared abstract. The class must be declared abstract too. The class cannot be instantiated.
A child class may implement some or all of the inherited abstract methods.
    If not all, it must be declared abstract.
    If all, it is not abstract and so can be instantiated.
If a class is completely abstract, we may choose instead to declare it to be an **interface**.

## Interface

**Interface**: a class with no implementation, it has method signatures and return types (guarantee capabilities)
all abstract methods, only signature of abstract methods
a checklist of methods (all connecting together), provide a service (the same service)

A class can be declared to "implement" an interface.
    This means it defines a body for every method.
    A class can implements 0, 1 or many interfaces, but a class may extend only 0 or 1 classes.

An interface may extend another interface.

[Example]: java.util.List        "To be a List, here are the methods you must support"

public class LinkedListQueue<T> implements
a variable representing a type

list in java are interface
interface list <E>

One class per java file, but class can be nested(not recommended).

ArrayList with more than one type in it

## Version Control (Git)

Backup and restore (because accidents happen)
synchronization (multiple people can make changes)
short term undo (that last change made things worse?)
long term undo (find out when a bug was introduced)
track changes (all changes related to a bug fix)
sandboxing

commit (I want to remember this)

Git
repositories was created by prof, clone your remote repository, work on files locally, add and commit changes to your local repository and push changes to the remote repository.

$ git add file1 file2..
    git add does not add files to your repo, it just prepares changes to add to a commit

$ git commit -m “adding feature x”

$ git status

A file can be in one of 4 states:

untracked: you have never run a git command on the file
tracked: committed
staged: git add has been used to add changes in this file to the next commit
dirty/modified: the file has changes that have not been staged

Note: use git status regularly…

Workflow:
start a project $ git clone <url>
normal work (after made some changes)
$ git status
$ git add file1 file2 file3
$ git commit -m “~~~”
$ git push // push changes to the remote repo

conflict in a file

## Week 4

### Interfaces

an interface called stack. Stacks have push, pop and isEmpty methods.
Implementation: array, ArrayList, LinkedList etc.

```
/** A LIFO data structure. */
public interface Stack {
  /** Add o to the top of this Stack.
   * @param o The object to be pushed.
   */
  void push(Object o);
  /** Remove and return the top item of this Stack.
   * @return the former top item of the this Stack.
   */
  Object pop();
  /** Return the top item of this Stack.
   * @return the top item of the this Stack.
   */
  Object top();
  /** Return whether this Stack is empty. */
  boolean isEmpty();
}
```

any class that want to implement this interface, has to have those methods.

interface: a description about how the … should be

All interface methods are automatically public.
Two ways to think of an interface:
①a guarantee to the client code that any class that implements the interface definitely has methods with these headers (and maybe other methods too)
②an obligation of the implementer, who must write these methods

```
/** A Stack with fixed capacity. */
public class ArrayStack implements Stack {
  /** The index of the top element in this Stack.  Also the number. */
  private int top;
  /** contents[0 .. top-1] contains the elements in this Stack. */
  private Object[] contents;
```

```
  /** An ArrayStack with capacity for n elements. */
  public ArrayStack(int n) {  contents = new Object[n];}
  /** Add o to the top. (Ignore that we might overflow.) */
  public void push(Object o) {  contents[top++] = o;}
/** Remove and return the top element of this Stack. */
  public Object pop() {  return contents[--top]; // What if top is 0?}
  /** Return true iff this Stack is empty. */
  public boolean isEmpty() {  return top == 0;}
}
```

Using a Stack:
You cannot create instances of interfaces (this is broken). Stack s = new Stack(15);
But you can write methods that used an interface.

```
/**
 * Fill a stack with the integers 0 to n - 1 (inclusive),
 * with n - 1 at the top.
 * @param the Stack to fill
 * @param n the number of integers to put into the stack
 */
public static void fill(Stack s, int n) {
  for (int i = 0; i != n; i++) {
    s.push(new Integer(i));
  }
}
```

That function will work with any class that implements Stack. Think of it as a service: it does work for anyone who needs their Stack filled with integers.

Note: casting down is required in some circumstances. casting up is not required, it happens automatically.

interface vs abstract class:
interface — no code, all signatures, nothing is implemented (provide a service)
interface and abstract class cannot be instantiated.

top ++; vs -- top;

In Java 8, there can be codes in interface, but everything has to be public, static () and final (no way to change the value).

Queue (intro to generics)
Queue Operations: enqueue, head, dequeue, size. All items in a queue must be the same type. (GENERICS)
<T> or <E> —> a reference type

```
/** A queue where all items must be of type T. */
public interface Queue<T> {
  /** Append o to me. */
  void enqueue(T o);
  /**
   * Return my front item.
   * Precondition: size() != 0.
   */
  T head();
  /**
   * Remove and return my front item.
   * Precondition: size() != 0.
   */
  T dequeue();
  /** Return my number of items. */
  int size();
}

/** A queue where all items must be of type T. */
public class LinkedListQueue<T> implements Queue<T> {
  /** The items in me.  Head is index 0, tail is index size() - 1. */
  private LinkedList<T> contents = new LinkedList<T>();
  @Override
  public void enqueue(T item) {
    contents.add(item);
  }
  @Override
  public T head() {
    return contents.get(0);
  }
  @Override
  public T dequeue() {
    return contents.removeFirst();
  }
  @Override
  public int size() {
    return contents.size();
  }
}

public class QueueDemo {
  public static void fill(Queue<Integer> queue, int num) {
    for (int i = 0; i != num; i++) {
      queue.enqueue(i);
    }
  }
  public static void main(String[] args) {
    // Here is where we decide which Queue implementation to use.
    Queue<Integer> queue = new LinkedListQueue<>();
    fill(queue, 10);
    System.out.println(queue);
  }
}
```

class Foo<T> introduces a class with a type parameter T.
<T extends Bar> introduces a type parameter that is required to be a descendant of the class Bar — with Bar itself a possibility. In a type parameter, "extends" is also used to mean "implements".
<? extends Bar> is a type parameter that can be any class that extends Bar. We'll never refer to this type, so we don't give it a name.
<? super Bar> is a parameter that can be any ancestor of Bar.

use generics to avoid casting.
casting is to cast the hierarchy down.

| | | |
|---|---|---|
| class a | public static field | public static method |
| class b | public static field | public static method |
| class c | public static field | public static method |
| | shadowing | overriding |

## UML (unified modelling language)

an extremely expressive language (2 main types of diagrams)
①structure diagram
classes interaction diagrams, components iteraction diagrams
(static view of the system)
②… diagram

Unified Modelling Language (UML)
use Class Diagrams to represent basic OO design

Data Members: name: type

Methods: methodName(param1: type1, param2: type2, …): returnType
Visibility: -private, +public, #protected, ~package private (default) (accessible modifier)
Static: <u>underline</u>
Abstract Method: *italic*
Abstract Class: italic or <<abstract>>
Interface: <<interface>>
Relationship between classes: inheritance → interface ⇢

[Example]:

| class Name | Person |
|---|---|
| Data Members | #name: String[]<br>#doc: String<br>#gender: String |
| Methods | +Person(name: String[], doc: String, gender: String)<br>+getName(): String[]<br>+setName(name: String[]): void<br>+getDob(): String<br>+setDob(dob: String): void<br>+getGender(): String<br>+setGender(gender: String): void<br>+toString(): String |

## Design Pattern

A design pattern is a general description of the solution to a well-established problem using an arrangement of classes and objects.
Patterns describe the shape of code rather than the details, they are a means of communicating design ideas and not specific to any one programming language.

Iterator vs Iterable
iterable: a class to store iterator

use iterator to avoid reorder the order stored in collection.

Note: no program involving main method!!! in the design pattern diagram

Assignment: to fix it up the given code, focus on design, pattern
Start with UML diagram!!!
—helper method
—inheritance

~makes code easier to read
~not too many classes, not too few classes
~easy to fix/adopt old code to new specifications later

command + N (abstract class existing in Object class)

Look-up Rules (Sub/Sup, when to parent class, when to child class)

Quiz 1 will be held in EX100 during lab time. It will last 45 minutes and cover:

- shadowing/overriding (when parent class and child class have variables with the same name and/or methods with the same signature)

- inheritance (how a child class obtains variables and/or methods without containing the code for them)
- types (subclasses of Object vs. primitive types)
- basics from the first two lectures (static/instance, accessibility modifiers, constructors, basic syntax, etc.)

types （primitive/object, all non-primitive are sub..sub..sub class of Object）

class 有默认的constructor 空的, 写了其他的constructor就没了默认的了

command + N                    new constructor

/* Field */        (attributes)
/* Constructor */
/* Method */

Getter and Setter method

psvm = create the main method shortcut

## Exception

In Java, an exception is an object.

—Write a catch block for each kind of exception that might happen, and code to deal with the exceptional situation.
—Let your method crash(!) and force the method that called yours to deal with it.

Throwable (Object that you can throw)
①Error <unchecked>: (Java will throw Error, we cannot throw Error, we can only throw Exceptions)
Something goes wrong

| | |
|---|---|
| AssertionError | unpreventable, unrecoverable (cannot deal with it) |
| OutOfMemoryError | unpreventable, unrecoverable |

②Exception<checked>: Something goes weird
IOException (file is deleted)                    unpreventable, deal-able (backup plan)

RuntimeException <unchecked> (ArithmeticException, ClassCastException)                preventable
Note: If I code properly, I can prevent RuntimeException.
Note: need to fix something in my code, never catch RuntimeException!!!

```
throw Throwable;           // throw an Exception
try {
       statements;
// The catch belongs to the try (like else and if)
} catch (ExceptionType1 parameter){
       statements;
} catch (ExceptionType2 parameter){
       statements;
}
finally{
       // optional, but finally block always run no matter what.
}
```

Note: the order of catch blocks matters! will ONLY execute one catch block.

```java
public class BankAccount {
  private double balance = 500;

  public double withdraw(double amount) throws
Insufficient {
    if(amount > balance) {
      throw new Insufficient();
    }
    balance -= amount;
    return balance;
  }

  public void safedraw(double amount) throws Insufficient
{
    try {
      this.withdraw(amount);
    } catch (Insufficient e) {
      System.out.println("Insufficient");
    } finally {
      System.out.println("Left " + this.balance);
    }
  }

  public static void main(String[] args) throws
Insufficient {
    BankAccount ba = new BankAccount();
    ba.safedraw(300);
    ba.safedraw(300);
  }
}

class Insufficient extends Exception {
  Insufficient(){}
}
```

```
Left 200.0
Insufficient
Left 200.0
```

```java
public class BankAccount {
  private double balance = 500;

  public double withdraw(double amount) throws
Insufficient {
    if(amount > balance) {
      throw new Insufficient();
    }
    balance -= amount;
    return balance;
  }

  public static void main(String[] args) throws
Insufficient {
    BankAccount ba = new BankAccount();
    ba.withdraw(300);
    ba.withdraw(300);
  }

}

class Insufficient extends Exception {
  Insufficient(){}
}
```

```
Exception in thread "main" Insufficient
        at BankAccount.withdraw(BankAccount.java:9)
        at BankAccount.main(BankAccount.java:28)
```

Throwable
Constructors:
Throwable(); Throwable(String message);
Methods:
getMessage();
printStackTrace();
getStackTrace();

Define your own exception:
choose a meaningful name so that it fits in the problem domain.
decide where in the exception hierarchy it belongs
decide how it fits into your regular code

public void m() throws MyException()
// tell the compiler that there might be a Exception, and let JVM check to make sure there is a try-catch block.
// if no “throws”, even if no try-catch, still can compile.
// unchecked: no “throws”, program will crash and stop.
method m()

two options: ?? difference?

try{x.m1();}
catch(SomeException ) {…;}

version control practice
check style (indentation, brackets, line between)

refactor (more readable, maintainable) — duplicated code (create a class to use or parent class to inherit)
javadoc (check javadoc.pdf as instruction) + other actually helpful comments
inheritance hierarchy diagram

Code Smell, Design Smell
    duplicated code (inefficient, one change leads to multiple modification required) —>
    large class (a class does one thing and does well. S"**O"**LID method of design)
    method too long —> helper method

**Sanity Check**: clone the entire repository again to make sure the PUSH actually goes to the remote repository.

Step 2: reformat (not quite bad)

Step 3: inheritance diagram

Step 4: check questions

Step 5 to N-2:

How to describe something?
message for step 18~~~

Step N-1: comments and Javadoc

Step N: add at least six new grid squared of rail to the track

## Week 5

Activity — Ticket Vendor Specification
class Ticket
class TicketManager
class Performance
class Account
class AccountManager

Class, Responsibility and Collaboration

CRC cards
—a tool and method for systems analysis and design
—part of the object-oriented development paradigm
—highly interactive and human-intensive
—final result: definition of classes and their relationships

class (name of class, one card for one class)
    an object-oriented class name
    include information about super and sub class
responsibility
    what information this class stores
    what this class does
    the behavior for which an object is accountable
collaboration
    relationship to other classes

which other classes this class uses

A CRC model is a collection of CRC cards. It specifies the Object-Oriented Design (OOD) of the software system.

Assignment 2: need to tell intellij which one is the source root!!!
send src to be the source root.

@Deprecated
Deprecated literally means *disapproved of*, but a more accurate translation would be *retired*. Deprecated means this method is still usable, but you should not use it. It will gradually be phased out. There is a new method to do the same thing.

When a class or method is **deprecated**, it **means** that the class or method is no longer considered important. It is so unimportant, in fact, that it should no longer be used at all, as it might well cease to exist in the future. The need for **deprecation** comes about because as a class evolves, its API changes.

Java Week 6: awt and swing

check java. …. logger.Scanner / Handler…

A design pattern is supposed to solve a problem.

events.txt -> input information (single line)
Phase II —> events.txt are scanner information

Do not consider salaries

Factory Design Pattern

A dependency relationship between two classes ("using" relationship) means that any change to the second class will change the functionality of the first.
Ex. class AddressBook depends on class Contact because AddressBook contains instances of Contact.
Ex. loggers, handlers, listeners

Hard Dependency: using the new operator inside the first class can create an instance of a second class that cannot be used nor tested independently (AVIOD HARD DEPENDENCY!)

Dependency Injection Design Pattern

## Regular Expressions/regex

Pattern in Characters

Describe a set of characters

Eclipse describes the Java naming conventions for variables this way ^[a-z][a-zA-Z0-9]*$
*: (a quantifier) repeat 0 or more times
^: **anchor**, beginning of a line (the pattern must start at the beginning of the string)
$: another **anchor**, end of a line (end of the string)
square brackets []: choose one of the characters listed inside.
This describes a pattern that appears in a set of strings. (any such string <u>matches</u> or <u>satisfies</u> the regular expression)

The entire string must be made out of letters and numbers with the first character being a lower case letter.

No anchors: [abc]C[a-e][24680]*[A-Z]
When this regex is applied to a string, it will find the substrings that match.
Example: cCaA, ABCcCcCa1A23 (contains substrings cCcC and cCa1A)

Other symbols
Space characters:
\s is a single space    \t is a tab character    \n is a new line character

Inside square brackets, ^ matches any character except the contents of the square brackets.
[^aeiouAEIOU] matches anything that is not a vowel.

Quantifiers:
* means zero or more, + means one or more, ? means zero or one
X{n} means X exactly n times, X{n,} means at least n times, X{n, m} means at least n but no more than m times

Sometimes we want symbols to show up in the String that otherwise have meanings in regular expressions. To "escape" the meaning of the symbol, we write a backslash \ in front of it.
A period . means any character.
To have a period show up in the string, write \.

abc123 matches [a-e][a-e].+    1.4 matches [0-9]\.[0-9]
.  any character    \d  a digit [0-9]    \D a non-digit [^0-9]
\s a whitespace char [tab \t \n \x0B \f carriage return \r]
\w a word char [a-zA-Z_0-9]    \W a non-word char [^\w]

\( \d{3} \) \d{3} - \d{4}    (416)399-2113

To repeat the same character twice, we use groups which are denoted by round brackets. Then we escape the number of the group we want to repeat.
^(([ab])c)\2\1$ matches acaca and bcbcb

Logical Operators
| means "or"
&& means the intersection of the range before the ampersands and the range that appears after.
[a-t&&[r-z]] would only include the letters r, s, t
If A is true X, else Y: (?(A)X|Y)
    If the regex A appears in the string, X must follow; otherwise, then Y must appear.

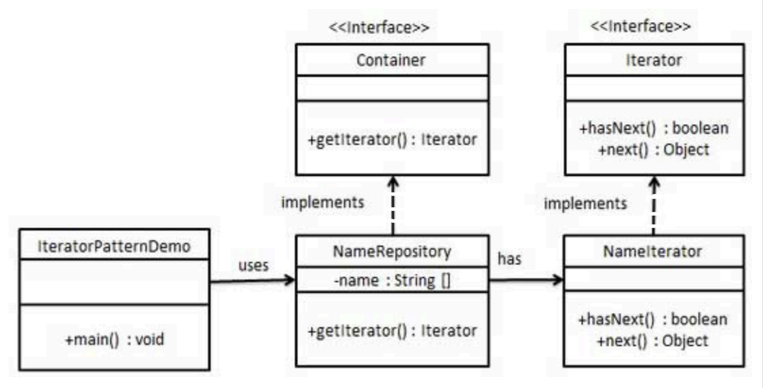
| | |
|---|---|
| Match a string containing two words that are the same. | .*\b(\w+)\b.*\1.* |
| Match a string containing 4 consecutive lowercase consonants. | [a-z&&[^euioa]]{4} |

## Design Pattern

### Singleton Pattern

### Iterator Pattern

To get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation



<<Interface>>

```
Iterator.java
public interface Iterator {
  public boolean hasNext();
  public Object next();
}


Iterable.java
public interface Iterable {
  public Iterator getIterator();
}
```

Create concrete class implementing the *Iterable* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

```
public class NameRepository implements Iterable {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {
         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {
         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

IteratorPatternDemo.java

```
public class IteratorPatternDemo {
   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
```

```
            System.out.println("Name : " + name);
        }
    }
}
```

Output

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

## Observer Pattern

there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically

Strategy
MVC
Dependency Injection
Factory Method

“anti-pattern”

## Practice Problem

1. Main Component of a class in java: field/variable, constructor, method
standard accessibility modifier: private
protected (default + subclass)
public
private
package-private: no modifier (package-private, default)
non-standard accessibility modifier: final, static, abstract

2.Method vs Constructor
Syntax
Java memory model

final:

## Interface Collection<E>

Superinterfaces: Iterable<E>
Subinterfaces: List<E>, Queue<E>, Set<E>, SortedSet<E>
Implementing Classes: ArrayList, LinkedList, Stack, Vector

## Graphical User Interface

JFrame, JPanel, JButton, BoxLayout, JCheckbox, ActionListener

## JUnit

reflection unit testing for private method

## Serializable

```
import java.io.*;
import java.util.ArrayList;

public class Student implements Serializable {
  private int num;
  Student(int studnetnum) {
    this.num = studnetnum;
  }
}

class StudentContainer implements Serializable {

  private ArrayList<Student> slist = new ArrayList<>();

  public void addStudent(Student s) {
    slist.add(s);
    System.out.println("There are " + slist.size() + " students!");
  }

  public static void main(String[] args) throws IOException {

    StudentContainer sc = null;

    // De-Serialization
    try {
      FileInputStream fileIn = new FileInputStream(new File("serialObj.txt"));
      ObjectInputStream in = new ObjectInputStream(fileIn);
      sc = (StudentContainer) in.readObject();
      System.out.println("Successfully Serialized!");
      in.close();
      fileIn.close();
    } catch (ClassNotFoundException e) {
      System.out.println("Employee class not found, but I create a class");
      sc = new StudentContainer();
      e.printStackTrace();
    } catch (FileNotFoundException e) {
      System.out.println("File not found, but I create a file");
      FileOutputStream fileOut = new FileOutputStream("serialObj.txt");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      sc = new StudentContainer();
      out.writeObject(sc);
      out.close();
    } catch (IOException e) {
      e.printStackTrace();
    }

    // Create new Student
    BufferedReader br = new BufferedReader(new InputStreamReader(system.in));

    int i = 0;

    while (i != -1) {
      System.out.print("Enter Integer:");
      try {
        i = Integer.parseInt(br.readLine());
        sc.addStudent(new Student(i));
      } catch (NumberFormatException nfe) {
        System.err.println("Invalid Format!");
      }
    }

    // Serialization
    try {
      FileOutputStream fileOut =
          new FileOutputStream("serialObj.txt");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(sc);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in serialObj.txt");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

public abstract/final/static void method ( ) {}

## Code Smells

### Application-level Smells

• Duplicated Code: identical or very similar code exists in more than one location.
• Contrived complexity: forced usage of overcomplicated design patterns where simpler design would suffice.

### Class-level Smells

• Large class: a class that has grown too large.
• Feature envy: a class that uses methods of another class excessively.
• Inappropriate intimacy: a class that has dependencies on implementation details of another class.
• Refused bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honoured by the derived class. (Liskov Substitution Principle)
• Lazy class / freeloader: a class that does too little.
• Excessive use of literals: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts where possible, to facilitate localization of software if it is intended to be deployed in different regions.
• Cyclomatic complexity: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
• Downcasting: a type cast which breaks the abstraction model; the abstraction may have to be refactored or eliminated.
• Orphan variable or constant class: a class that typically has a collection of constants which belong elsewhere where those constants should be owned by one of the other member classes.
• Data clump: Occurs when a group of variables are passed around together in various parts of the program. In general, this suggests that it would be more appropriate to formally group the different variables together into a single object, and pass around only this object instead.

### Method-level Smells

• Too many parameters: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way.
• Long method: a method, function, or procedure that has grown too large.
• Excessively long identifiers: in particular, the use of naming convention to provide disambiguation that should be implicit in the software architecture.
• Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious.
• Excessive return of data: a function or method that returns more than what each of its callers needs.